Value Sensitive Programming Language Design

NICHOLAS MARQUEZ
School of Computer Science
Georgia Institute of Technology

A programming language is a user interface. In designing a system's user interface, it is not controversial to assert that a thoughtful consideration of the system's users is paramount, indeed consideration of users has been a primary focus of Human-Computer Interaction (HCI) research. General-purpose programming language design has not had much need for disciplined HCI methodology because programming languages have been designed by programming language users themselves. But what happens when programmers design languages for non-programmers? In this paper we claim that the application of a particular design methodology from HCI – Value Sensitive Design – will be valuable in designing languages for non-programmers.

ADVISOR:
CHARLES L. ISBELL
School of Computer Science
Georgia Institute of Technology

INTRODUCTION

A programming language is a user interface. In designing a system's user interface, it is not controversial to assert that a thoughtful consideration of the system's users is paramount. Though there is a large body of research from the Human-Computer Interaction (HCI) research community studying just how best to consider a system's users in the design of its interface there is little history of applying any of these methodologies from HCI to the design of general-purpose programming languages. Ken Arnold has argued that, since programmers are human, programming language design should employ techniques from HCI (Arnold 2005). While there has been some work in applying HCI to the design of languages for non-programmers, for example, for children's programming environments (Pane et al. 2002), general purpose programming languages have not suffered much from a lack of HCI methodology in their design because programming languages have been designed by programmers, for programmers. In other words, programming language design has not had much need for disciplined HCI methodology because programming languages have been designed by programming language users themselves. But what happens when programmers design languages for non-programmers? How does the language designer know which design decisions to take? We claim that these questions can and should be answered with the help of a disciplined application of design methodologies developed in the HCI community.

We are designing a language for non-programmers who use computational models in the conduct of their non-programming work, in particular social scientists and game developers who write intelligent agent-based programs. Agent-based programming has, as one of the primary abstractions, "agents" who interact with each other and their environment asynchronously, maintain their own state, and are generally analogous to individual beings within an environment. In designing this language, we believe that working closely with our intended users is crucial to the development of tools that will meet their needs and be adopted. To guide our design interactions with our users we are applying the Value Sensitive Design (VSD) (Friedman et al. 2006; Le Dantec et al. 2009) methodology from HCI. In this paper we give a short description of VSD and discuss how it may be applied to the design of our programming language. This work is currently at an early stage, and our understanding and application of VSD is evolving. Nevertheless we believe that the application of HCI methodologies in general, and VSD in particular, will be extremely valuable in the development of languages and software tools that are intended for non-programmers, that is, professionals for whom programming is an important activity but not the primary focus of their work.

In the next section we provide a brief description of Value Sensitive Design (VSD), then we propose a way of applying VSD to programming language design and conclude with a discussion of how we are applying it in our own language design project.

VALUE SENSITIVE DESIGN

In this section we briefly describe VSD as detailed in Friedman, et. al. (2006). We begin with their definition of VSD:

> Value Sensitive Design is a theoretically grounded approach to the design of technology that accounts for human values in a principled and comprehensive manner.

In this context, a value is something that a person considers important in life. Values cover a broad spectrum from the lofty to the mundane, encompassing things like accountability, awareness, privacy, and aesthetics –

anything a user considers important. While VSD uses a broader meaning of value than that used in economics, it is important to rank values so that conflicts can be resolved when competing values suggest different design choices.

VSD employs an iterative, interleaved tripartite methodology that includes conceptual, empirical, and technical investigations. In the following sections we describe each of these types of investigations.

Conceptual Investigations

We think of conceptual investigations as analogous to domain modeling. In conceptual investigations we specify the components of particular values so that they can be analyzed precisely. We specify what a value means in terms useful to a programming language designer. Conceptual investigations may be done before significant interaction with the target audience takes place. As is characteristic of VSD, however, conceptualizations are revisited and augmented as the design process proceeds in an iterative and integrative fashion.

An important additional part of conceptual investigation is stakeholder identification. Direct stakeholders are straightforward – they are the people who will be writing code in your language using the tools you provide. However, it is important to consider indirect stakeholders as well. For example, working programs may need to be delivered by your direct stakeholders to third parties – these third parties constitute indirect stakeholders. The characteristics of indirect stakeholders will implicate values that must be supported in the design of your language. If the indirect stakeholders are technically unsophisticated, for example, then the language must support the delivery of code that is easy to install and run.

Empirical Investigations

Empirical investigations include direct observations of the human users in the context in which the technology to be developed will be situated. In keeping with the iterative and integrative nature of VSD, empirical investigations will refine and add to the conceptualizations specified during conceptual investigations.

Because empirical investigations involve the observation and analysis of human activity, a broad range of techniques from the social sciences may be applied. Of all the aspects of VSD, empirical investigation is perhaps the most foreign to the typical technically focused programming language designer. However, as computational tools and methods reach deeper into realms not previously considered, we believe empirical investigations are crucial to making these new applications successful.

Technical Investigations

Technical investigations interleave with conceptual and empirical investigations in two important ways. First, technical investigations discover the ways in which users' existing technology supports or hinders the values of the users. While these investigations are similar to empirical investigations, they are focused on technological artifacts rather than humans. The second important mode of technical investigations is proactive in nature: determining how systems may be designed to support the values identified in conceptual investigations.

APPLYING VSD TO PROGRAMMING LANGUAGE DESIGN

In this section we discuss the ways in which we are applying VSD to the design of a programming language. First we discuss the language itself and the target audience of our language

AFABL: A Friendly Adaptive Behavior Language

AFABL (which is the evolution of Adaptive Behavior Language) integrates reinforcement learning into the programming language itself to enable a paradigm that

we call partial programming (Simpkins et al. 2008). In partial programming, part of the behavior of an agent is left unspecified, to be adapted at run-time. Reinforcement learning is an area of machine-learning focused on the technique of having an agent perform actions in its environment which optimize (usually maximize) some concept of a reward. Using the reinforcement learning model, the programmer defines elements of behaviors – states, actions, and rewards – and leaves the language's runtime system to handle the details of how particular combinations of these elements determine the agent's behavior in a given state. AFABL allows an agent programmer to think at a higher level of abstraction, ignoring details that are not relevant to defining an agent's behavior. When writing an agent in AFABL, the primary task of the programmer is to define the actions that an agent can take, define whatever conditions are known to invoke certain behaviors, and define other behaviors as "adaptive," that is, to be learned by the AFABL runtime system.

We are designing AFABL for social scientists and other agent modelers who are not programmers per se but who employ programming as a basic part of their methodological toolkit. We also hope to encourage greater use of agent modeling and simulation among practitioners who currently do not use agent modeling, and among agent modelers who would write more complex models if given the tools to do so more easily. Since these kinds of users have very different backgrounds from programmers it is important to understand their needs and values in designing tools intended for their use. We believe that VSD will be one methodological tool among perhaps many that will help us understand our target audience and truly incorporate their input into the design process. In the next section we discuss how this process is taking place in the design of our language.

Using VSD in the Design of AFABL

We are working with two different groups who are currently using agent modeling in their work. The first group, the OrgSim group (Bodner and Rouse 2009), is a team of industrial engineers who are studying organizational decision-making using agent-based simulations as well as other more traditional forms of simulations. The OrgSim group wants to model the human elements of organization in order to create richer, more realistic models that can account for human biases, personality, and other factors that can be simulated only coarsely, if at all, using traditional simulation techniques. The second group is a team of computer game researchers creating autonomous intelligent agents that are characters in interactive narratives (Riedl et al. 2008; Riedl and Stern 2006).

Both of these groups are using the most advanced agent modeling language available to them: ABL (A Behavior Language) (Mateas and Stern 2004). ABL was created in the course of a Computer Science PhD by a games researcher to meet his needs in creating a first-of-its-kind game, an interactive drama. ABL was not designed with the help of or goal of assisting non-programming expert agent modelers. Naturally, both groups have met with difficulty in using ABL. By using VSD in working with these groups we hope to meet their needs with AFABL.

Conceptual Investigations of Agent Modelers

As described earlier, conceptual investigations yield working definitions of values that can be used in the design of technological artifacts – in our case the AFABL programming language. In our conceptual investigations thus far we have identified several values, whose conceptualizations as we currently understand them are listed below.

- Simplicity. Here simplicity has two essential features. First, AFABL must be consistent in its design, both internally and in the extent to which it exploits the users' current knowledge of programming. Internal consis-

tency means that when users encounter a new language construct for the first time, they should be able to apply their knowledge of analogous constructs they already know. External consistency means that AFABL should use programming constructs that users already know form other languages and require users to learn as few new language constructs as possible.

• Power. A language is sufficiently powerful if it allows its programmers to reasonably and easily write all the programs they want to write in the language. If a language makes it hard to write certain types of programs, then those programs will usually not be written, thus limiting the scope of use of the language. Naturally, power trades off with simplicity, but simplicity at the expense of essential power is unacceptable to our target audience. In the design implications section below we discuss strategies for dealing with the power versus simplicity issue.

• Participation. Our user communities are eager to contribute to the design of AFABL and to its documentation and development of best practices. We welcome this participation and believe that it will positively impact adoption, both with the users with whom we are already working and new users that will be influenced by our early adopters. VSD directly supports and encourages this participation in the design process.

• Growth. The language we develop and the theoretical models of intelligent and believable agents that we employ today may not be the last words. It is important that AFABL be able to accommodate new models and applications.

• Modeling Support. A modeling tool imposes a structure on the way an agent modeler thinks about agents. AFABL should do so in a helpful way, if possible, but certainly not hinder particular ways of thinking about agents.

Empirical Investigations of Agent Modelers
Solving a problem requires an understanding of the problem. The problem in our case is the experience of agent modelers in using the computational tools at their disposal. To understand the problems agent modelers face and their desiderata for computational tools, we are joining their teams and using their existing tools alongside them. In doing so we hope to gain an appreciation for the goals of their work, the expertise they bring to the task, and the difficulties they have in using existing tools to accomplish their goals. We hope to gain a level of empathy that will help us develop a language and tools that will meet their needs very well.

Technical Investigations of Agent Modelers
What do they already use? How do their existing tools support or hinder their values? What technology choices do we have at our disposal to support their values? These are the kinds of questions we address in technical investigations. In our case, there is a rich tapestry of software tools already in use by our users. These tools include virtual worlds — simulation platforms and game engines — and editing tools for the programs they currently write. Some of these tools are essential to their work and some may be replaced with tools we develop. One overriding value that stems from our users' existing tool base is interoperability. Any language or tool we develop must support interoperability with their essential tools.

Implications of Values on Progamming Language Design

We are already familiar with many values supported by the general purpose programming languages we use. C supports values like efficiency and low-level access. Lisp supports values like extensibility and expressiveness. Python supports simplicity. In this section we discuss how some of the values we identified above may impact the design of our language.

Interoperability It is essential that our language and tools support interoperability with the virtual worlds currently in use. In our technical investigations we have found that the simulation platforms and game engines in use support Java-based interfaces, and many of them run on the Java Virtual Machine (JVM). Since these projects also use ABL, they have existing code bases that use the ABL run-time system and bridge libraries that enable communication between ABL agents and virtual worlds. These technical investigations lead us to the following design decisions for AFABL:

• AFABL will run on the JVM. Currently, we are planning to implement AFABL as an embedded domain specific language (EDSL) written in the Scala programming language (Odersky et al. 2008). This will allow us to interoperate well with Java programs and ABL while providing advanced language features in the design and implementation of our language.

• AFABL will use the ABL run-time system. While we have decided to depart from the syntax and language implementation of ABL, the agent model and run-time system of ABL represents a tremendous amount of valuable work that we wish to build on, not reinvent. Additionally, using the ABL run-time system will allow us to make use of the existing bridges between ABL agents and virtual worlds.

Simplicity and Power Simplicity and power often oppose each other when taking design decisions, so we discuss these values here together. We hope to maximize both power and simplicity with the following language features:

• Provide a simple consistent set of primitives and syntax while providing expressive power through first class functions, closures, objects, and modules. Languages like Ruby and Python can be used by programming novices as well as expert programmers who use advanced expressive features such as iterators, comprehensions, closures, and metaprogramming. We intend to employ the same strategies in the design of AFABL.

• Feel free to make presumptions / Optimize for the common case — A great majority of the time, modelers will be using similar methods and approaches. There should be as little friction between the modeler's thoughts and the compiler's input as possible. Being able to make sound prejudgments about the programming language's users and the patterns of programming they exhibit is key to opening a whole class of optimizations and simplifications that can help both the user and the compiler. In the context of AFABL, this means that we very much need to evaluate our design at every step with our target userbase and should employ, e.g., VSD in doing so.

• Do not assume anything / Keep uncommon and unforeseen cases possible — Only close off the language where it would create great disparity of future implementation or for necessary optimizations. In the latter case (should the common case be in use) the alternate, optimized, but less extensible implementation can be used. One should not outright assume anything about the user (because this would restrict future ways in which the language could be used), and should take care to properly document and account for any presumptions. We must be sure to focus AFABL not too much towards our VSD-driven presumptions, lest we unintentionally restrict the ease of use of the language for other types of modelers and users.

Participation Our users have expressed strong interest in contributing to the design, documentation, and practices of AFABL. To accommodate our users' desire for participation we anticipate the following features of AFABL:

• Iterative language development. By designing AFBL around a small set of primitives, we hope to get the lan-

guage into the hands of users early in its development. That way users can experiment with the language and provide feedback throughout its development. Put another way, AFABL will be developed with agile software development practices.

• User-accessible documentation system. Many languages already provide programmers with the means to automatically generate documentation from source code. Many language communities also provide user-accessible documentation systems, such as Wikis and web forums, whereby users can share their knowledge and contribute directly to the documentation base of the language. We will employ similar mechanisms for AFABL.

Growth New theories of agent modeling and new virtual worlds will be created in the future. To accommodate these changes, we will design AFABL for growth in the following two ways:

• Support new run-time systems. The ABL run-time system represents a particular way of modeling behavioral agents. It may be possible to support new agent theories by connecting AFABL with new run-time systems.

• Support the full range of operating system and JVM intercommunication. By providing a full set of intercommunication mechanisms, such as pipes, sockets, file system access, and JVM interoperability, AFABL should be able to accommodate new virtual world environments.

CONCLUSION

In this paper we have taken the position that design methodologies from the HCI research community can be of great benefit in the development of programming languages. Among the design processes we are employing, we have singled out Value Sensitive Design and described how it can be used in the design of programming language and tools for a non-traditional population of programmers, in our case agent modelers like social scientists and game designers.

ACKNOWLEDGEMENTS

I wish to thank David Roberts for suggesting the use of Value Sensitive Design, and Doug Bodner and Mark Riedl for allowing us to participate in their projects and their help in designing AFABL.

REFERENCES

Arnold, K. Programmers are people, too. Queue, 3(5):54–59, 2005.

Bodner, D.A. and Rouse, W.B. Handbook of Systems Engineering and Management, chapter Organizational Simulation. Wiley, 2009.

Friedman, B., Jr., Kahn, P.H., and Borning, A. Human-Computer Interaction in Management Information Systems: Foundations, chapter 16. M.E. Sharpe, Inc, NY, 2006.

Le Dantec, C.A., Poole, E.S., and Wyche, S.P. Values as lived experience: Evolving value sensitive design in support of value discovery. In Proceedings of the ACM Conference on Human Factors in Computing Systems (CHI 2009), Boston, MA, USA, April 2009.

Mateas, M. and Stern, A. Life-like Characters. Tools, Affective Functions and Applications, chapter A Behavior Language: Joint Action and Behavioral Idioms. Springer, 2004.

Odersky, M., Spoon, L., and Venners, B.. Programming in Scala. Artima, 1 edition, 2008.

Pane, J.F., Myers, B.A., and Miller, L.B.. Using hci techniques to design a more usable programming system. In Symposium on Empirical Studies of Programmers (ESP02), Proceedings of 2002 IEEE Symposia on Human Centric Computing Languages and Environments (HCC 2002), Arlington, VA, September 2002.

Riedl, M.O. and Stern, A. Believable agents and intelligent scenario direction for social and cultural leadership training. In Proceedings of the 15th Conference on Behavior Representation in Modeling and Simulation, Baltimore, Maryland, 2006.

Riedl, M.O., Stern, A., Dini, D., and Alderman, J. Dynamic experience management in virtual worlds for entertainment, education, and training. In International Transactions on Systems Science and Applications, Special Issue on Agent Based Systems for Human Learning, volume 4(2), 2008.

Simpkins, C., Bhat, S., and Isbell, C.L., Jr. Towards adaptive programming: Integrating reinforcement learning into a programming language. In OOPSLA '08: ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, Onward! Track, Nashville, TN USA, October 2008.